(4)

RADC-TR-88-132, Vol III (of four)
Final Technical Report
June 1988

CRONUS, A DISTRIBUTED OPERATING SYSTEM: Interim Technical Report No. 5

BBN Laboratories incorporated

R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, K. Lebowitz and R. Sands

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss AFB, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-132, Volume III (of four) has been reviewed and is approved for publication.

APPROVED: Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans and Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific doucment require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): Report No. 5991	5. MONITORING ORGANIZATION REPORT NUMBER(S): RADC-TR-88-132, Volume III (of four)
6a. NAME OF PERFORMING ORGANIZATION: BBN Laboratories Incorporated; 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION: Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code): 10 Moulton Street, Cambridge MA 02238	7b. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Rome Air Development Center; 8b. OFFICE SYMBOL (If applicable): COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F30602-84-C-0171
8c. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 63728F; PROJECT NO. 2530; TASK NO 01; WORK UNIT ACCESSION NO. 26

11. TITLE (Include Security Classification): CRONUS, A DISTRIBUTED OPERATING SYSTEM: Interim Technical Report No. 5

12. PERSONAL AUTHOR(S): R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, K. Lebowitz and R. Sands

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Oct 84 TO Jan 86	June 1988	66

16. SUPPLEMENTARY NOTATION: N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Distributed Operating System, Heterogeneous Distributed System, Interoperability, System Monitoring & Control, Survivable Application
12	07		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This is the final report for the second contract phase for development of the Cronus Project. Cronus is the name given to the distributed operating system (DOS) and system architecture for distributed application development environment being designed and implemented by BBN Laboratories for the Air Force Rome Air Development Center (RADC). The project was begun in 1981. The Cronus distributed operating system is intended to promote resource sharing among interconnected computer systems and manage the collection of resources which are shared. Its major purpose is to provide a coherent and integrated system based on clusters of interconnected heterogeneous computers to support the development and use of distributed applications. Distributed applications range from simple programs that merely require convenient reference to remote data, to collections of complex subsystems tailored to take advantage of a distributed architecture. One of the main contributions of Cronus is a unifying architecture and model for developing these distributed applications, as well as support for a number of system-provided functions which are common to many applications

(over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ OTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Thomas F. Lawrence	22b. TELEPHONE (Include Area Code): (315)330-2158; 22c. OFFICE SYMBOL: RADC (COTD)

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

Block 19 (Cont'd)

This report consists of four volumes:

Vol I - CRONUS, A DISTRIBUTED OPERATING SYSTEM: Revised System/Subsystem Specification

Vol II - CRONUS, A DISTRIBUTED OPERATING SYSTEM: Functional Definition and System Concept

Vol III - CRONUS, A DISTRIBUTED OPERATING SYSTEM: Interim Technical Report No. 5

Vol IV - CRONUS, A DISTRIBUTED OPERATING SYSTEM: CRONUS DOS Implementation

Table of Contents

Appendices

Appendix A: The Architecture of the Cronus Distributed Operating System

Appendix B: Programming Support in the Cronus Distributed Operating System

Appendix C: RFC 947: Multi-network Broadcasting within the Internet

1. Introduction

This report is an interim technical report for the Cronus Distributed Operating System Implementation project. It covers the period between October 1984 and May 1985.

1.1. Project Overview

The objective of this project is to extend the current Cronus Distributed Operating System (DOS) implementation*, completing the basic functionality for supporting distributed system demonstration software; to extend the testbed environment with additional hosts and tools to support the development and evaluation of Air Force applications; and to begin to establish a second testbed cluster on-site at RADC. The overall function of the DOS is to integrate the various data processing subsystems into a coherent, responsive and reliable system which supports development of distributed command and control applications. The development work for this contract is broken down into the following areas:

Area	SOW Item
VAX-UNIX Integration	4.1.1.2.1
SUN Workstation Integration and Use	4.1.1.2.2
Resource Management	4.1.2
Survivability	4.1.3
Reconfiguration Support	4.1.3.2
Tool Integration	4.1.4.1
Application Development Support	4.1.4.2
RADC Cluster Support	4.1.5
Cluster Maintenance	4.1.6

In addition to the development work, a report on how new hosts and their resources are integrated into a Cronus cluster will be written (SOW 4.1.1.2.3). A demonstration of the system and its capabilities will be presented at the end of the contract period (SOW 4.1.8).

1.2. Organization of this Report

The following sections describe the progress that has been made so far on each item in the statement of work. In the appendices, we include recently written papers that describe, in detail, aspects of the work done on the Cronus system architecture, on language support for distributed application development, and on network support for Cronus.

---

*For a description of previous Cronus development, see *CRONUS, A Distributed Operating System: Phase 1 Final Report*, R. Schantz, et al. BBN Report No 5885, January 1985.

2. Integration of New System Hardware

Under the previous Cronus development effort we established an initial demonstration environment. It consisted of three types of hosts: 68000 Multibus microprocessor systems running the CMOS operating system as Generic Computing Elements (GCEs), and two types of utility hosts, BBNCC C70 running UNIX and DEC VAX 11/750 running VMS. The GCEs are small dedicated-function computers of a single architecture but varying configurations. They provide specific Cronus services, such as file managers and terminal access points. The utility hosts provide the program development and application execution environment for Cronus. Most of our development activities were centered on C70 Unix because of its rich set of development tools and the ease of developing new software afforded by the UNIX environment.

We have added support for VAX-UNIX and the SUN Workstation. The VAX-UNIX represents an evolution of the existing Cronus UNIX support to a new hardware base. The SUN Workstation represents a new class of Cronus host which was described in the Cronus hardware architecture but not previously supported.

2.1. VAX-UNIX

VAX-UNIX is presently supported on both the VAX 11/750 and 11/785. The hardware base for these implementations are currently owned and operated by the BBN Computer Systems Division to supply timesharing support for the company. The larger of the machines, the 11/785, typically supports 40-50 users. Cronus applications run concurrently with non-Cronus timesharing workload on these hosts.

The VAX supports a large virtual address space under the Berkeley 4.2BSD release of UNIX. The operating system for the C70, our other UNIX based utility host, does not support virtual memory and is based on the earlier Version 7 Unix from Bell Laboratories. In addition to virtual memory support, the 4.2BSD provides many new features and languages, and improved interprocess communication and I/O facilities, and better overall performance.

The VAX-UNIX system serves to replace the C70 as a hardware base for future DOS and related application development. The VAX family of computers is widely accepted, with a large installed hardware base, which increases the likelyhood of finding existing machines to integrate into Cronus.

The VAX-UNIX systems support the Cronus operation switch, all managers, including the file and catalog manager, all the application development tools and all Cronus user commands. We have also made modifications to speed development of Unix based utilities for accessing Cronus files. We have modified the standard C compiler libraries so that file I/O routines will invoke the appropriate Cronus operations whenever a Cronus file name is given. This has allowed us to simply recompile many UNIX file utilities, such as *cp*, *cat*, *grep*, and *diff*, and the text editors *emacs* and *vi* to produce versions that access both Unix and Cronus files. In some cases, minor modifications were required to the source programs.

2.2. SUN Workstation Integration and Use

The SUN Workstation is a 68000 Multibus system based on the SUN microprocessor board developed at Stanford University. It includes a high-resolution raster graphics display with a mouse input device and a window based user interface. The system supports virtual memory under a version of Berkeley 4.2BSD UNIX, essentially the same as the VAX-UNIX described above. The SUN is representative of the trend toward powerful, single-user computers with high performance graphics capabilities that make feasible man-machine interfaces of significantly higher quality than those possible on time-shared mainframe computers communicating with terminals over slow, bit-serial links.

We have installed two Sun Model 120 Workstations, each with a 130 megabyte Winchester disk drive and 2 megabytes of primary memory. These systems offer enough power for use as workstations or for use as utility hosts for program development by 2-3 users performing typical development tasks. The workstations support the Cronus operation switch, all managers, including the file and catalog manager, all the application development tools and all Cronus user commands. The sources for these Cronus programs are essentially identical to the sources used for the VAX system.

We are also developing a workstation based Monitoring and Control System (MCS) for the Cronus cluster. Beyond the major focus on issues of monitoring and control for a distributed system, we are exploring the use of graphics facilities supported by the workstation as an operator interface. An MCS system consists of three parts: the MCS operator interface, the data collector and the event reporting system.

The first version of the user interface is based on the BBN Graphics Editor, a subsystem previously developed by BBN to serve as an environment for constructing graphical interfaces. Based on object-oriented programming techniques and implemented on the Sun workstation, the Graphics Editor permits the interactive composition of graphical diagrams, or views. These diagrams are dynamic control panels that can be connected to data sources and sinks and used to graphically control and display the state information. Using this system we have produced views that summarize cluster host status, the status of each of the services, and the status of the managers for each service.

The data collector collects and monitors status information about the managers and the objects they manage. The collector periodically invokes the *report status* request to retrieve the information from the managers. This information can be recorded for later review. The values can be displayed using the operator interface, either for a particular point in time or to view the trend over a period of time, Values can be monitored so that the operator will be warned when a particular value crosses an operator specified threshold.

The event reporting system is used to alert the operator when irregular events occur. For example, when a manager is restarted it submits an event report to inform the MCS. These event messages are displayed on the MCS console and can also be recorded for later review.

3. Resource Management

As a distributed system architecture, Cronus faces a number of resource management issues not present in non-distributed architectures. In this phase of development we have focused on the binding of a request from a client to a particular resource manager for those resources which are available redundantly. Redundancy comes in two forms: replicated objects and replicated managers. In both cases the selection of an object manager to provide the given service is an important resource management decision.

The general approach to resource management in Cronus is to individually control the management of the classes of objects which make up the system. This approach extends resource management concepts to the abstract resources developed by applications. In addition to this system resource management, application and system interface code can, if they desire, control resource management decisions to incorporate larger purviews such as implementing a policy which tries to optimize the use of collections of different objects types used in a specific context.

We have implemented mechanisms that allow resource management at two levels: by the client submitting the request and by the collection of managers responsible for a each type. The client may collect status information about the available managers using the *report status* request and then direct the invocation of an operation to a particular manager. The client specifies in the request that the operation must be performed by the specified manager; no resource management decisions will be made by the manager itself in this case. If the operation cannot be performed by the selected manager it will refuse the request and the client must choose a different manager to continue. Normally, requests do not identify a particular instance and the type managers make resource management decisions. The managers collect status information from their peers using the *report status* request and then forward the client request to the manager best suited to perform the operation. The manager to which the message is forwarded will process the request and reply directly to the client that originally issued the request.

To experiment with resource management and to test the mechanisms, we have modified the primal file manager to implement a resource management policy for creating new files. The mechanisms work as follows. An initial request to create a new file is routed at random to any available file manager based on response to a locate operation. When a primal file manager receives a file create request it checks the local space usage and processor load. If either of these parameters exceeds operator selected thresholds, the file manager will not process the request itself. Instead, using status collected from the other managers it will choose the one it considers to be best suited to perform the operation. It then forwards the request, along with the appropriate access control rights, to the selected manager for processing. The policy parameters that guide the selection can be set by the operator through the MCS operator interface or by users invoking simple commands available elsewhere in the cluster.

4. Survivability Enhancements and Reconfiguration Support

A primary goal of the Cronus architecture is survivability in the face of system component failures. In the C^2 environment it is especially important to provide continuous availability of key applications despite system failures. There are two aspects of survivability which the Cronus architecture addresses: the availability of the system and its services over a relatively long period of time and the survivability of the applications which it runs. Application survivability is dependent not only on sustaining the application itself and the abstractions it presents to its users, but also on sustaining the resources on which it depends for its computational support. The object oriented approach taken in Cronus gives us an appropriate general approach to these problems. The objects and functions needed to sustain a computation must be made survivable.

Our approach is to support replication and reconfiguration through the manager development tools provided for application development support. In this way we can use common techniques for both system and application objects, and make these techniques conveniently available to application developers. We will be experimenting with multiple and customized approaches to replication support via the manager development tools. Managers developed with the tools use access routines to a standard object database. These library routines perform the coordination and duplication needed to update all the copies of an object maintained by the managers of that type. When a manager is restarted, initialization routines communicate with other managers of the same type to update the new manager's database, which may have fallen out of date while it was unavailable.

Using these tools, we have implemented a replicated authentication manager. This was done both to experiment with the use of the tools and because replication of the authentication function is fundamental to system survivability. We are currently running two instances of the authentication manager. When both are running, they share the load of login requests. When one goes down, the remaining one handles all the requests and updates the other when it returns to service. We will be applying the survivability mechanisms to other system objects in the near future.

5. Distributed Application Development Support

We feel that the object metaphor may be extended into the application domain. That is, one develops a new application by first defining the types of objects involved and then the operation protocols they follow. Cronus has been designed to support a commonality of structure between system and application components, including the use of common mechanisms, particularly those designed to aid distribution, resource allocation, and reliability, and common development tools. In this section we describe our initial steps toward developing a distributed application development environment.

5.1. Development of New Types

So far, increasing support for more easily adding new types has involved work in two areas: making the system aware of the new type, and providing support for the automated development of a skeletal manager. We maintain a type database that stores a specification for each type and the operations that may be invoked on each type. In addition, we support tools which given a specification of the types implemented by a manager, provide the skeleton code for the manager. This skeletal code provides request dispatching, multi-tasking support, access control, resource allocation and replication. For each operation the code supports access control and unpacks the message into a suitable data structure, checking to make sure all required parameters have been provided. The tools also implement generic operations such as locate and access control list modifications. A subroutine library provides the underlying support and provides a database for storing the objects maintained by the manager. Only implementation of the individual operations is left to the developer. The developer can override or replace the code provided by the tools when necessary. To complement the convenient development of the type manager, we also support automatically generating synchronous client interface subroutines for invoking each operation.

5.2. Software Distribution Manager

Software distribution in a distributed development environment, though seemingly simple, often becomes an extremely complicated, time consuming, and error-prone task. We believe much of this is due to the volume of data required to describe the distribution requirements in terms of "(*file,site*)" pairs, as is often done by developers. Our most important desire was to provide a simple abstract model to the user and to limit the amount of information the user has to understand and manipulate. Our approach is to group files with identical distribution requirements into *packages*. Each package lists the files it contains and the sites to which those files are to be distributed. This representation is more natural to the developer of large applications, since such a user will normally think in terms of collections of files composing a distributed application or subsystem, and this representation provides a much more concise description of the distribution requirements than listing the "(*file,site*)" pairs.

We felt that the distribution process should be controlled by a logically-centralized manager process, rather than independently from a variety of client programs. This has the benefit of limiting knowledge of the implementation of packages to one program, and of minimizing the interface requirements at the user access point since the user need only be able to invoke a single Cronus operation.

Similarly, we wanted to minimize the amount of software required at each site-bearing host. The current implementation requires only one instance of the controlling part of the software distribution manager for the entire network, although more may be employed for load balancing and to provide survivable functionality. The addition of a new site bearing host only requires the development of Constituent Operating System (COS) Interface Manager on the new host, a service normally provided anyway.

We also felt that, for the initial version, it was important to keep the system conceptually simple, particularly with respect to assuring and verifying consistency between sites. This led to the notion that developers should explicitly distribute updates after they are confident that the files are internally consistent, in contrast to using a daemon process that regularly looks for changes at the designated primary site and distributes updates, when appropriate, to ensure that all instances are continually consistent.

Finally, we thought it important to provide adequate access controls. For example, maintenance of the lists of files and sites are independently access controlled to reflect the differing roles of software developers who modify the implementations and system administrators who determine the ultimate location of services.

To ease the implementation, and provide a test vehicle for earlier work, we decided to implement our solution exclusively using Cronus facilities. The Software Distribution Manager was constructed using the manager development tools, and invokes operations on other managers using the automatically-generated program support library subroutines. The manager is not dependent on the contents and semantics of the files in a package. They may be source files, language processor header files, shell scripts, or, when distributed between hosts of the same type, binary executable and library files.

5.3. Integration of Editors, Compilers and other Tools

Cronus is both a base operating system for supporting distributed applications and an environment for developing these applications. One important aspect of supporting software development in a distributed environment is a distributed file system. A distributed file system is useful only to the extent that there are tools which can utilize the distributed file system. An initial step toward making Cronus more useful for software development is to provide a set of development tools which utilize Cronus functionality. Such tools include editors, compilers and linkers.

At the outset, we have chosen to adapt existing tools to the Cronus environment whenever possible, rather than developing tools specifically tailored for the Cronus environment to gain immediate functionality To reduce the effort required to adapt existing tools, we have modified the subroutine libraries for the VMS, C70 Unix and Vax Unix systems. These "trap" libraries invoke Cronus operations whenever a file name specifies a Cronus file. Otherwise, they behave as they did before modification: performing the operations on VMS or UNIX files.

The VAX-UNIX trap library was developed during the first part of this contract and has been used to produce several UNIX based file utilities as mentioned in an earlier section. We plan to convert the SUN libraries in the same way when sources become available. These library routines intercept file operations and invoke Cronus operations whenever a Cronus file is specified. Otherwise the routines act as they did before modification.

5.4. Distributed Access to Constituent Operating System File Systems

Through Cronus, it is also desirable to gain remote, distributed access to directories and files maintained by a Constituent Operation System (COS). This allows remote access to mailboxes, bulletin boards, on-line manuals and other data that are common to several systems but normally required either duplicates to appear on all the systems or require the client to connect to the system where the data is stored. We have implemented a manager, called the *COS Interface Manager*, which provides access to directories and files stored on the COS. Registering a COS file or directory with this manager returns a Cronus Unique Identifier (UID) that can be later used to manipulate it remotely as a Cronus object. The UID of the COS file or directory is commonly stored in the Cronus catalog, providing a global symbolic name for it. In many cases, Cronus users need not be aware of whether a particular catalog entry refers to a Cronus primal file or a COS file. Thus, the Cronus utilities, such as *display* and *ldir*, work with COS files and directories as they do for Cronus objects. The COS Interface manager is a step in the gradual evolution between completely independent host systems and a completely integrated distributed system.

6. RADC Cluster Support

An important part of demonstrating the applicability of Cronus in the C^2 environment, evaluating its capabilities, and successfully transferring DOS technology is the installation and operation of a Cronus DOS cluster at RADC. Doing this will provide valuable experience in transporting Cronus to another environment and seeing how well it can be operated and used by a different user community. The Cronus cluster at RADC will be gatewayed to the DARPA Internet so that it can be accessed remotely from the cluster at BBN. This will allow both remote operation and monitoring of the RADC cluster and experimentation with inter-cluster operations in the Cronus DOS.

We have been assisting RADC with the selection of the hardware configuration for the Cronus cluster. We have already submitted specifications for the hardware configuration. In order to facilitate installation and operation of the RADC cluster, our major guideline in the selection has been compatibility with the BBN cluster, at least in terms of the types of machines and operating systems supported and the underlying local network. We are preparing a cluster installation report that details how to install Cronus once the cluster hardware has been installed.

7. Cluster Maintenance

In addition to general maintenance and bug fixing we have made several improvements to Cronus to upgrade its operational capabilities and performance. These enhancements include the following.

We have begun work to extend the implementation of Cronus to span multiple physical networks. A broadcast repeater allows managers to locate objects and other managers that are connected to other networks. This works by propagating broadcast requests between two local networks via the internet. See Appendix C for a more detailed discussion of the issues to consider

when building a broadcast repeater and for a description of the architecture of the repeater we have built.

Large messages are now supported and use a mechanism that exploit the length of the message to reduce overhead. Rather than sending large messages as a sequence of small messages, each routed through the Cronus kernels of both the client and server, the Cronus kernels and program support library routines establish a direct TCP connection between the client and server. The message is then transmitted across this connection without the need for either kernel to be involved further. See *Cronus System/Subsystem Specification**, section 6.4: "*IPC Implementation*" for additional details.

We have also reduced the amount of time spent locating managers for a particular type and for an instance of a particular object. The Cronus kernel now maintains an object address cache where it records the results of locate requests. Since most clients, once they have referenced a particular type or object, will make additional references to that type or object, the additional locate requests can be satisfied from the contents of the cache. This eliminates the delays and traffic that would arise from exchanging network messages to satisfy the addition locate requests. Support in the program support library ensures that the cache will be updated if its contents have become invalid because an object has been moved to a manager has become unavailable. In such cases, the locate request will be issued and the cache will be updated to reflect the new location.

8. Constituent Operating System Integration Guidelines

Integrating new hosts into Cronus is one of the long term objectives for the system. Having already performed a number or such integration tasks we have begun to prepare a document describing the host capabilities that are necessary or desirable for participation in the Cronus environment.

9. Papers and Technical Articles

9.1. SOSP Papers

Two papers, included as Appendix A and Appendix B, have been submitted to the review committee for the December 1985, Symposium on Operating System Principles. The first of these, *The Architecture of the Cronus Distributed Operating System*, describes the overall architecture of Cronus and details the design of key components of the system. The second paper, *Programming Support in the Cronus Distributed Operating System*, presents our approach to the problem of distributed application development, describes the features of Cronus that support this development, and illustrates how Cronus facilitates development using a Cronus object manager as an example.

---

*Cronus System/Subsystem Specification, R. Schantz, et al. BBN Report 8554, Revision 1.4, June 1984.

9.2. Broadcast Repeater RFC

The paper included as Appendix C has been distributed as an Arpanet RFC 947. It describes the extension of a network's broadcast domain to include more than one physical network through the use of a broadcast packet repeater.

Appendix A

The Architecture of the Cronus Distributed Operating System

Richard E. Schantz
Robert H. Thomas
Girome Bono

BBN Laboratories
10 Moulton Street
Cambridge, Massachusetts 02138
(617)-491-1850

Table of Contents

Appendix A

The Architecture of the Cronus Distributed Operating System

1. Introduction

The Cronus distributed operating system is intended to promote and manage resource sharing among interconnected computer systems. Its major purpose is to provide a coherent and integrated system based on clusters of interconnected heterogeneous computers which supports the development and use of distributed applications. Distributed applications range from simple programs that merely require convenient reference to remote data, to collections of complex subsystems tailored to take advantage of a distributed architecture. Among the main contributions of Cronus is a unifying architecture and model for organizing these distributed applications, and tools for their development in the form of system functions which are common to many applications. The Cronus system is itself an example of this type of organization and uses the support mechanisms in its implementation.

Cronus is a third-generation distributed operating system. Our earliest experiences with first-generation distributed systems [RSEXEC, SBS, 5SOSP] provided insight into the issues of network-based interprocess communication, message passing systems, and distributed operating system functionality in a homogeneous environment. Our second-generation distributed system [NSW, NSWI, White] gave further experience in the areas of heterogeneous system components, functional specialization, language-oriented approaches toward distributed systems, and many aspects of supporting the operational use of distributed computing systems.

In 1981 we began work on Cronus. Our immediate aim was to capitalize on our previous experience and bring it up to date to include experiences of other related projects, significant advances in both hardware and software technology, and the changing scope of the problems being considered appropriate for distributed system architectures. Since we had only a vague notion of any intended applications, flexibility to adapt to a wide variety of potential uses was important. The idea was that if it was relatively "easy" to build distributed applications, people would find ways to utilize this capability. Almost four years later, the diversity of potential uses for Cronus seems to have partially validated that approach. We have had a version of Cronus running in our laboratory for over a year, providing system integration and various system services on a variety of hardware and operating system bases.

This paper describes the overall architecture of Cronus, and details the design of key components of the system. A companion paper [Gurwitz] describes a continuation of this work in the areas associated with programming Cronus applications. Other papers in progress will cover the design and implementation of various functional and support areas in depth.

2. Project Overview

2.1. Strategic Assumptions

The orientation of the Cronus system is derived from a number of key observations based on previous experience. Like Watson [Watson], we believe that one of the reasons there are so few significant distributed applications is that to date, development of distributed systems has required the application developer to spend too much attention on the details associated not only with networks and communication but also with heterogeneity, synchronization, etc. Further, each application developer needs to solve problems such as naming and access control, which are common to most distributed applications, for each new application context. If application developers are provided with "off-the-shelf" solutions to these common problems, more distributed systems are likely to emerge.

The second observation is that what people are looking for from distributed systems is multi-dimensional and encompasses a large problem area. There does not seem to be a single most important aspect. Rather, there are collections of problems and desireable system properties that all seem to suggest a distributed system architecture as a solution. This observation led us to stress a comprehensive system architecture under which we could preplan to address many aspects of distributed system technology simultaneously.

The third observation is that the system developed must evolve. This evolution is likely to take a number of forms. One form is evolution of the design, since the problem is too large to be addressed all at once. Another form of evolution is recognition that parts of the system design and implementation will be reconsidered and possibly redone as the hardware, underlying system support, applications and system concepts change over time. Experience has also shown that it is difficult to displace ingrained patterns of user behavior with even the best of new technology. A more prudent approach seems to be to accommodate current functionality side by side with evolving new functions.

A fourth observation is that building a distributed operating system in a heterogeneous environment is an exercise in handling complexity. Therefore, the structuring of the overall system into manageable units is an important issue.

2.2. Objectives

With these basic assumptions as background, we set out a number of specific design and implementation objectives for Cronus. The primary objective was to establish a comprehensive distributed system architecture and design for integrating a collection of different computer systems into a coherent and uniform computing facility which serves as a base for developing distributed applications. Within this facility the system would provide uniform, coherent mechanisms for various functions

including communication, access control, naming, and data storage and retrieval. Furthermore, this computing facility should exhibit the following properties:

- o survivability of system functions
- o scalability of system resources
- o global management of system resources
- o ability to substitute system (hardware) components for each other
- o convenient operation of the collection of systems.

The approach to developing Cronus has been to establish a general framework for addressing these objectives, and then to elaborate the design in each of these areas. We believe that many of the properties desired for the Cronus functions are also desirable for applications, and that an "open" system model, where applications can be constructed using the same mechanisms that support the system is a good approach. Cronus development is a continuing activity. The effort to date has concentrated on developing an extensible distributed system architecture, establishing an initial Cronus hardware testbed facility, designing and implementing a model for host-independent access to system resources, and establishing systemwide uniformity in a variety of DOS functional areas. We have also begun to address issues of survivability, resource management, and monitoring, and control. In a companion effort [Berets] we are performing test and evaluation of the current system by developing a collection of interrelated, multihost applications. In this paper we are reporting those aspects of Cronus that have achieved a degree of stability from everyday use in our laboratory testbed.

2.3. System Environment

Cronus operates in an environment made up of interconnected computer communication networks [Postel]. This internet environment includes both geographically distributed networks, which span tens to thousands of miles, and local area networks, which span distances of up to a mile or two. From an architectural point of view, it is useful to think of this environment as being composed of clusters of host computers, where the hosts within a cluster are localized and are typically under a single administration.

A cluster may include hosts on several networks, and several clusters may exist on the same network. Performance considerations will generally lead to clusters that consist of hosts on a single local area network or on a few local networks interconnected by means of high-performance gateways. Therefore, although a cluster is a logical rather than a physical concept, it is our feeling that they will tend to be aligned with local area networks.

Cronus currently operates in a cluster defined by one or more local area networks. Extensions to multi-cluster architectures are currently being designed. The principal elements in a Cronus cluster include:

1. A set of hosts upon which Cronus operates.

2. One or more high-performance local area networks which support communication among hosts within a Cronus cluster.

3. An internet gateway which makes the cluster part of the large internet environment by supporting communication between cluster hosts and hosts external to the cluster.

The Cronus host set is a heterogeneous collection of hosts which can be divided according to function into three broad classes:

1. Hosts dedicated to providing Cronus functions.

   The functions the hosts provide include data storage, user authentication, catalog management, device control, and terminal access. The hosts which support these functions are called Generic Computing Elements (GCEs). GCEs are inexpensive, dedicated-function computers of a single architecture but varying configuration. Each GCE provides one or more basic Cronus functions. Because they are dedicated to Cronus, it is possible to control and optimize the performance and reliability of the Cronus services supported on GCEs. In particular, Cronus can be the native operating system on GCE hosts.

2. Utility and application hosts.

   These hosts support some Cronus functions which may also be supported by GCEs, but their primary role is to support user applications. The utility and application hosts include a variety of machines with differing architectures. They are often mainframe hosts which may serve a number of users simultaneously. The software necessary to integrate them into Cronus runs as an adjunct to rather than a replacement for the hosts' primary constituent operating system (COS), since this software generally is or supports software that determines why the host is part of the configuration in the first place. Hosts can be included in Cronus with varying degrees of system integration, with some directly supporting only limited subsets of the services defined by the Cronus environment. Included in this category are various general-purpose utility hosts supporting commonly accessible services such as database management or high-speed multiprocessor architectures.

3. Single user workstations.

   Workstations are powerful, dedicated computers which provide substantial computing power and graphics capability to a single user. They are used both to provide user access to Cronus

and for their ability to run applications. They differ from application hosts in that they support a single user and from terminals in that they offer significant computational resources.

3. System Architecture

The basic system organizing principal underlying Cronus is an abstract object model [Jones, Almes]. With this model all system activity can be thought of as operations on objects organized into classes called types. Entities such as processes or directories are examples of Cronus types. Each object is under the direct control of a manager process on some host in the Cronus cluster. The resources of the system are cast as types, with manager processes resident on each host in the cluster which supports instances of that type. A type manager on a Cronus host manages all objects of that type which reside on the host. The collection of managers for a given type collectively manage the resource represented by that type for Cronus.

The underlying structure of Cronus, which is largely hidden from client processes, consists of the primitives and mechanisms for delivering operations invoked by clients to the appropriate manager for an object, and delivering the results, if any, of operations back to the invoking client. Location transparency and dynamic binding are two important characteristics of the network orientation of Cronus that are reflected in its object model. Support for location transparency permits operation invocation to be completely independent of the sites of the client and the object being accessed. A given object can be accessed in precisely the same manner from any point in the system. The dynamic binding of client requests to appropriate object managers supports maximum flexibility afforded by the network context. Some objects can migrate to serve as the basis for reconfiguring the system, while others are replicated to support survivability. The approach to scalability is through integrating additional hosts and managers for a resource type. Global resource management is achieved through the cooperation of the managers for a given resource. System monitoring and control functions are achieved by monitoring and controlling the behavior of the various object managers. Supporting these attributes through the object model means that we can easily tailor solutions to the particular resource type.

There are three interrelated parts to the Cronus software architecture:

1. The Cronus kernel, which supports the basic elements of the object model.
2. A group of basic object types, along with the object managers which implement them.
3. User interface and utility programs.

Every host integrated into Cronus must support a kernel. The Cronus system kernel includes a object-based message passing facility, supporting the invocation of operations on objects. The kernel itself provides the notions of host objects for monitoring and control purposes, and the process objects for supporting Cronus managers and application programs. A Cronus library provides a standardized interface for invoking operations on objects, including conversion to and from a standard data exchange format for interprocessor communication (masking the heterogeneity within the cluster). Other basic object types and managers used as building blocks supporting Cronus application software include:

- o User identity objects, called *principals,* and objects which are collections of principals, called *groups,* used to support user authentication and the Cronus access control mechanism. These objects are managed by an authentication manager.
- o Directory objects and directory managers that implement a global symbolic name facility used to catalog other Cronus objects.
- o File objects and file managers that support a distributed filing system.
- o Device objects and device managers that support the integration of I/O devices into Cronus.

User interface and utility software run as applications to support user command interfaces and to aid in operating the system. Because object access is host-independent, application software can be run with the same results on any host in the cluster that supports that type of program.

Cronus is based on the idea that the user sees essentially no difference between "system" services and "application" services. The Cronus object model provides an extensibility mechanism to support application development. It includes a set of rules for building and accessing new types of objects, which spell out the methods for integrating new object managers. Cronus treats all types uniformly, in accord with its object model. Application programmers can use the object model for the standard access paths it provides to existing objects and functions, or they can use Cronus facilities to create new objects and new type managers. The Diamond [Diamond] multi-media message system is an example of an application which has successfully applied the Cronus distributed application paradigm in its design and implementation.

A basic Cronus system is augmented by the resources which are available on the variety of constituent systems which populate a given cluster. At the heart of the Cronus concept is the availability of its functions to all Cronus applications through host transparent invocation. A Cronus configuration consists of a collection of hosts, each of which supports some of the resources of the system with access to the other resources made available through operations invoked on them.

3.1. Objects and Operations in Cronus

The definition of an object in Cronus is tailored to the distributed nature of the system. Special emphasis is placed on allowing efficient access to objects without detailed information about their current physical location.

All Cronus objects have several components:

1. A Unique Identifier (*UID*). A UID is a fixed-length structured bit string guaranteed to be unique over the lifetime of the system. It serves as a global low-level name for a particular object, used to reference the object from anywhere in the system. It consists of a unique number or *UNO* and an *Object Type* field. The UNO guarantees uniqueness and incorporates the host upon which the object was created. The type serves to classify the object. Although ultimately all references to objects are through UIDs, Cronus implements a symbolic name space through its distributed catalog function which provides a mapping between user-defined symbolic names and object UIDs to facilitate user references to objects.

2. A Set of Operations. Processes may perform operations on an object by sending request messages to the object's *manager*. An object manager is a process or set of processes responsible for maintaining and manipulating an object. By convention, all managers are responsible for performing several common operations on their objects to support various systemwide functions such as access control, resource monitoring etc. Managers also perform any number of object-specific operations.

3. An Object Descriptor. This is data associated with the object. It is maintained by the object's manager. It consists of several required fields and any number of object-specific fields. Some of the generic operations are defined for accessing object descriptors. Cronus achieves a consistent system model largely from the uniform integration and handling of these object attributes and from the common operations which apply to all objects.

A process may declare itself a manager of one or more object types. A service is typically supported by a set of functionally equivalent and cooperating manager processes distributed on various hosts of the system.

A useful property of type managers is that they may be accessed by simply knowing the object type that they are responsible for. A special UID called the *generic UID* for the given type is provided to make such access possible. Generic UIDs are used for creating new objects and for initiating status probes to monitor the service represented by the type using. Generic UIDs for object addressing effectively provides a way to multicast communication with the collection of managers supporting the type.

Every Cronus object has a UID. Each object manager maintains a record of UIDs for objects it manages in a UID Table. When a manager creates an object it creates an entry for the new object in its UID Table. Each manager's UID Table defines a part of the UID name space. The entire Cronus UID name space is defined by the union of the UID tables of all the object managers. Thus, there is no single identifiable catalog of UIDs supporting the UID name space. Rather, the Cronus UID name space is implemented in a distributed fashion, with each object manager responsible for implementing part of it.

A key element of the object model is the Cronus kernel, which supports communication between client and object manager processes. The kernel is message-oriented, and it supports object-oriented addressing. The message routing portion of the kernel is often referred to as the *operation switch*. When an operation is invoked on an object, the operation switch delivers the operation (in a message) to the appropriate object manager. Messages corresponding to operations are sent as messages addresssed to the objects. The object addressed is the operand, and the message data contains the operation and any additional parameters necessary to specify the operation. When the manager for the object receives the message, it performs the operation requested. Responses are sent as messages from object managers to requesting client processes.

A consideration unique to the distributed environment is the location of resources. It is often impossible to guarantee the availability of certain hosts in a configuration; yet it is desirable to use them when they are available. Also, useage patterns vary with time and increased load. Cronus provides support for these specialized problems by defining objects which may be moved from one host to another, or which may be replicated on several different hosts, and by supporting a dynamic binding procedure for accessing these objects.

When invoking an operation a process need not specify the host where the addressed object resides. [1] To deliver the message, the kernel must determine the appropriate host using the object UID. In general, three somewhat different classes of objects are accessed through the kernel. These are:

1. Primal Objects

   These are forever bound to the host that created them.

2. Migratory Objects

   These are objects that may move from host to host as situations and configurations change.

3. Replicated and Structured Objects

   These are objects which have more internal structure than a single "atomic" object. An

[1] However, provision is made for a program or user through his command interface to optionally specify a particular host for the object or operation.

example is a reliable (replicated) file which has a number of identical primal files as its constituent parts.

3.2. Object Location and Message Routing

Primal objects are the simplest kind of object. The kernel routes an invoke on a primal object by using the host of origin from the object's UID, and delivering the message to the appropriate manager process on that host. Migratory objects may move from one host to another. The object managers are responsible for much of the mechanism necessary to support migrating objects. They implement, by convention, manager-to-manager protocols for moving objects, and forwarding for misdirected messages to previously migrated objects. The operation switch binds invocations to migratable objects by first broadcasting a *Locate* request to all potential managers of an object. Locate is one of the common operations implemented for all objects in the system. The correct manager, if it is available, answers the request, and the message is delivered to it. As an optimization, the current locations of recently accessed non-primal objects is cached. A Locate operation on the generic type-UID is used to find an available manager for that type.

Replicated objects are the most complex. A replicated object is maintained simulataneously by a number of manager processes on different hosts. Its manager processes keep copies of the object data, which they synchronize by means of manager-to-manager comunication. Mech· isms for synchronizing replicated objects may vary for different object types. Invocation binding is handled in the same way as for migratory objects, except that any of the available managers of a replicated object may answer locate requests. Currently the first one to respond is chosen for the operation invocation, although other algorithms could be used to promote load sharing or enhance reliability.

3.3. Message Passing Core

Process-to-process messages form the basis of all Cronus operations. A Cronus operation in the simplest case consists of a request message from a client to an object, and a reply message from the manager of the object sent back to the client. A complex operation may involve many subrequests to various managers and replies to each of the requests. Still other more complex patterns which involve message forwarding are also supported.

The primitive operations available to Cronus processes are *Send* , *Invoke* , and *Receive* . Send is used to send a message directly to a process. Invoke is used to perform an operation on an object. An Invoke is delivered to the manager of a given object, based on its type. Receive is used to obtain the next message. There is a subtle distinction between the Invoke and Send primitives. The client need not know the specific identity of a manager to direct a request to it using Invoke. The target of an Invoke is the object UID. It is the IPC mechanism that routes the message to the object's manager. Although processes are themselves objects, for efficiency Send operations invoked on process objects (usually responses to invocation requests) are routed directly to the addressed process, not the process manager. Additionally, the separation of the Invoke operation from the Receive that often follows it allows for complex asynchronous operations and optimizations involving parallel execution. In particular, managers often Invoke sub-operations while they are simultaneously being available to start new operations. [See Gurwitz for more details].

3.4. Access Control in Cronus

All client access to Cronus objects is subject to access control. The goals of the access control mechanisms are: to prevent unauthorized use of Cronus and Cronus objects: to preserve the integrity of the system; and to provide users a uniform view of access control for all Cronus resources, services and objects.

The basis of access control in Cronus is the ability of the Cronus kernel to reliably deliver the identity of the invoker of an operation to the receiver of the message. The recipient can then decide on the basis of the sending client's identity whether or not to perform the operation requested on the particular Cronus object. For this to be a useful basis for access control there must be a means for reliably associating authorizations with clients. Mechanisms are required to establish bindings between client processes and authorities, and for object managers to determine the authority bindings for client processes.

Ultimately, most activity within Cronus is the result of requests initiated by users. Users are represented internally to Cronus by objects of the basic type "principal". The authority bindings are, therefore, a correspondence between client processes and principals. System elements, such as object managers, also execute under the authority of a principal.

To control access, the identity and authority of the principal associated with the client process that requests an operation is checked prior to performing the operation. Access control in Cronus involves two things: determining the identity of the principal requesting the operation (identification authentication): and determining whether the principal has been authorized to perform an operation on the particular

object (authorization verification).

Cronus uses access control lists to support authorization verification. In its simplest form, an access control list (ACL) is a list of principals that serves to limit access to an object for a particular action to those principals on the list. This simple idea is extended in two ways:

1. The UID for a group of principals may appear on an ACL [Grapevine]. This makes it possible to authorize a group of principals rather than authorizing each individually. (Like principal. group is a basic Cronus type.)

2. A set of rights is associated with each UID on an ACL. There is a right associated with each operation defined for the object. Each right in the set represents authorization to perform one or more particular operations. This makes it possible for an ACL to selectively control access to an object on a per-operation basis, and for the rights to be customized for each new type.

When a user attempts to start a Cronus session, a process is allocated for him. The authority-binding for that process cannot be established until the user demonstrates that he is an authorized user. The login operation involves an authentication dialogue between the user and Cronus through which the user supplies a name (of a principal) and a password (for that principal object). The login operation is implemented by the authentication manager who manages principal objects. The authentication manager currently runs on a single host in the cluster. We are in the process of making principals replicated objects for survivability. If the name and password are valid. the set of groups to which the user belongs is computed from a list of group UIDs maintained as part of the named principal object. Since groups can contain groups, this is a transitive closure computation [Robertson]. The user's principal UID is combined with the result of the computation to form a set called the access group set (AGS). The AGS is then bound to the authenticating process through its process manager. This ensures the availability of the AGS binding whenever the process is able to initiate operations. Processes subsequently created by an authenticated process inherit the AGS of the creating process.

To perform an access control check for an operation on an object, the manager for the object first determines the AGS binding for the client process. The identity of the client process is known to the manager because its UID is delivered by the Cronus Operation Switch along with the message that requests the operation. The manager obtains the AGS of the client by invoking the BindingOf operation on the client process object. After obtaining the AGS, the manager can perform the access control check by comparing the AGS with entries on the ACL. When new objects are created. they are given an access control list which may be initialized under client control through type specific initial access control lists which are stored with directory and principal objects. An initial access control list can be set based on where the object is cataloged or who is creating the object. There are generic operations and associated user commands which apply uniformly to all objects for manipulating access control lists.

3.5. The Cronus Symbolic Catalog

Cronus supports two systemwide name spaces for referencing objects. At a relatively low level there is the name space of object UIDs supported by the Cronus kernel and object managers. At a higher level there is a symbolic name space for Cronus objects. The Cronus catalog supports the symbolic name space. The catalog provides a mapping between the symbolic names that people use to refer to objects and the UIDs that are required to actually access the objects.

Within Cronus access to an object is initiated in one of two ways:

1. Directly through the UID name space.

   The accessing client process has the UID of the desired object and invokes an operation upon it. The operation switch delivers the requested operation to the appropriate object manager. The object manager consults its fragment of the UID Table to access the object as necessary to perform the requested operation.

2. Through the symbolic name space.

   The accessing process has a symbolic name for the object. In this case, the catalog is searched for a catalog entry for the name, using a name lookup operation. If an entry is found, the UID for the named object can be obtained from it and used to access the object as in (1) above.

An object may have zero, one, or more symbolic names. When an object is given a symbolic name, an entry for the name is made in the catalog, and when the name for an object is removed, its entry is removed from the catalog. Symbolic names are location-independent that is, a name for an object is independent of its host location within Cronus, and a name that refers to an object may be used regardless of the location within Cronus from which the reference occurs.

The symbolic name space is structured hierarchically as a tree, much like the UNIX and Multics file name hierarchies. In Cronus any object may be given a symbolic name. For example, principals and groups have symbolic names which are managed by the Cronus Catalog. Leaf nodes in the name space tree represent Cronus objects which have symbolic names, and non-terminal nodes correspond to directories. Directories are objects which taken together form the catalog.

The Cronus Catalog is implemented in a distributed fashion by a collection of catalog managers on several Cronus hosts. A directory is a collection of catalog entries. The unit of dispersal across the collection of catalog managers is the directory.

In general to interpret a symbolic name, the lookup operation follows a path through the name space tree. With no restrictions on the dispersal of the catalog that path could pass through many different directory sites. It is desirable for performance and reliability reasons to limit the number of sites that are involved in a lookup operation. Two useful restrictions on the dispersal of the catalog [ELAN] are to require that:

1. The catalog structure for entire subtrees below a certain cut (the "dispersal cut") through the catalog tree be stored within a single site. A subtree that is rooted at the dispersal cut is called a "dispersal subtree".

2. The catalog structure above the dispersal cut be replicated on all catalog hosts. The structure above the dispersal cut is called the "root portion" of the hierarchy.

The effect of these restrictions is that any lookup operations require at most two catalog sites.

We replicate the root portion of the catalog hierarchy to increase its availability and distribute its load. We maintain a copy at each catalog site, making each functionally equivalent for operations on the root portion, and allowing many lookup operations to be completed at a single site.

To ensure that an object is accessible symbolically whenever the site that stores the object is accessible, we maintain a secondary symbolic access path to objects. The secondary access path is supported by maintaining copies of catalog entries at each object-managing host for objects managed at that host. In situations when appropriate parts of the catalog are unavailable, the secondary symbolic access path is be used.

A catalog manager currently runs on all application hosts in the Cronus testbed.

3.6. Host and application integration.

When a new application host is integrated into Cronus there are a range of integration possibilities which have different cost-versus-degree of integration trade-offs. When a host is integrated with minimal effort, little more than a communication path between the host and the rest of Cronus will be present. The host will be able to obtain Cronus services through the communication path, but its own resources may be inaccessible to external processes. Further effort can be devoted to integrate the host more fully into Cronus by supporting local instances of current functions or by developing new ones. Here, the object model allows sufficient flexibility to apply transparent remote access to existing host resources or to fabricate new application resources. In particular, very high level objects, which require most or all of a host's resources to support, can be defined for hosts which do only one thing particularly well, or are otherwise hard to modify.

The granularity of objects is an important issue in an environment where communication costs are significant. Processing seems to be the least expensive computing resource and will probably remain so. Controlled communication is relatively expensive. An important aspect of our system design is the ability to incorporate processing in the interface to all data storage components. An object manager represents a placeholder for arbitrary, user-specifiable pretransmission processing on the data it manages. By sending the request for processing to the data instead of moving the data to the accessing site for processing we can take advantage of the processing-versus-communication cost tradeoffs. In some environments, selecting an appropriate high-level abstract resource and tailoring its abstract operations to minimize reliance on high-bandwidth communication represents a complementary approach to optimizing communication support for achieving acceptable performance.

4. System Implementation

4.1. Testbed Configuration

The current Cronus testbed configuration consists of about 15 hosts, including BBN C70 minicomputers running a version 7 UNIX derivative, a DEC VAXes running VMS, and 4.2 BSD, SUN workstations running 4.2BSD UNIX, and Motorola 68000 microprocessor systems as GCE's interconnected by a 10MB Ethernet. Other VAXes (running 4.2BSD and VMS) on a second Ethernet connected to the first via an ARPA standard gateway are also integrated into our cluster. Each of these systems supports a Cronus kernel and an individually configurable complement of Cronus object managers from among file, directory and authentication managers. Each of the application and workstation hosts can serve as Cronus access points. A variety of resources specific to the application hosts have been easily integrated into the Cronus environment, and are in daily use by project staff. A number of new Cronus applications are in the design stages now.

4.2. Implementing Cronus System Components

Demonstrating that Cronus could operate on a set of heterogenous architectures and systems was critical among the implementation goals. There were two conflicting issues in planning the implementation for our testbed hosts. On one hand we wanted an efficient implementation. Ideally, initiating an invocation would be as efficient as a system call in a conventional system. On the other hand, because we needed to provide implementations for a number of different hosts and anticipate many more, we wanted the implementation to be as independent of any particular system context.

To minimize the time and implementation effort we emphasized code portability over efficiency for at least the first version. On application hosts most of the Cronus kernel is implemented outside the COS kernel to maximize its portability. The major issue we faced in installing a Cronus kernel on an application host side by side with its COS is establishing communication between clients and the operation switch. It is conceptually easy but in practice difficult to put the Cronus kernel directly inside the COS kernel. Instead, we utilize an available local host IPC mechanism to support client-kernel communication. For example, on some UNIX systems we use an intra-host datagram service, while on VMS we use mailboxes. These interfaces can be upgraded in the future, but with significant cost. Current performance levels are adequate and do not warrant such steps at this time. To the contrary, we have reaped enormous benefit from the ability to produce new versions of the kernel quickly, easily, and frequently. A native implementation of the Cronus mechanisms, such as on the GCE, is not dependent on local host IPC.

We chose C as implementation language for the initial Cronus components because of its availability and standardization. We antcipate application code to be developed in a variety of languages. The original version of the Authentication Manager was written in Pascal.

A great deal of portability has been achieved. Most managers and application programs port after only recompilation from a single set of sources. The kernel is the most host-dependent component, but it too is approximately 85% source code compatible on all the systems.

4.3. Network Support

Message passing in a network environment is fundamental to Cronus. To achievethe ability to substitute components in the network, Cronus accesses the local network capabilities indirectly through an interface called the Virtual Local Network (VLN). The VLN embodies an abstraction of local network capabilities. It is based on the network-independent IP datagram standard [DOD] augmented with broadcast and multicast capabilities. The current Cronus development cluster uses an Ethernet. By implementing Cronus using the VLN interface, it is possible to replace the Ethernet easily with any local network that provides the basic transport services. Cronus has already been ported to a Pronet Ring network base with only network device drivers needing recoding.

Messages may be sent either reliably or with minimal effort. Normally, messages are passed between hosts using a reliable transmission protocol. Reliable messages involve positive communication level acknowledgement and retransmission. Messages which are passed for informational purposes often don't need guaranteed delivery. Minimal-effort messages avoid the overhead of these features. The Cronus implementation is based on the standard DoD transport protocols IP and TCP. IP is used to send minimal effort datagrams. All broadcast messages are minimal effort. The current implementation of reliable message transmission uses a full duplex reliable TCP connection between peer operation switches for transporting Cronus operation invocations and replies. These connections can remain open indefinitely while there is communication between the hosts. To avoid problems with scalability and a dynamically

changing host environment, these connections can be activated and deactivated at any time by either side.

Cronus messages can be of arbitrary size. [Rashid] To avoid burdening the operation switches with buffering large amounts of data and avoid the extra operation switch overhead for each data segment, large messages are transported over a direct process-to-process TCP connection. The distinction between small and large messages is hidden from client processes by Cronus library routines. Messages may be sent and received all at once or in pieces. The size of the chunk of data manipulated is independently selected by the sender and receiver. Large messages of indefinite size form the basis for interprocess stream communication.

The Cronus implementation is based on a layered architecture. This provides the opportunity to use only a subset of Cronus for a specific or limited application, and to replace individual parts of the implementation easily with alternative but equivalent implementations, should the need arise for optimization purposes.

Cronus makes extensive use of broadcast facilities provided by its communication base to locate object managers and objects dynamically. It is often true that a selected collection of hosts to be integrated into a Cronus system is not limited to a single LAN. In order to extend Cronus utilities across several local area nets and allow hosts which are not on a LAN to participate fully in Cronus, it is necessary to extend the broadcast mechanism beyond the local area network. Since Cronus uses IP internet addressing, regular (i.e. non-broadcast) messages can already traverse multiple networks without special handling.

To support multinetwork broadcast (and multicast) services we have implemented a broadcast repeater which listens for broadcast messages on its LAN, and forwards the packets to other LANs. The broadcast repeater system serves as a transparent medium for relaying broadcast packets from one LAN to another, and also for forwarding broadcast packets to off-LAN hosts. Only one repeater system is needed per network. The broadcast repeater system can selectively forward broadcast traffic according to a variety of fields in the message to control the volume.

4.4. Related Work

Related work and a summary of a number of future directions for the Cronus effort, are discussed in a companion paper [Gurwitz].

Acknowledgements

This work has been supported by the Rome Air Development Center, under contracts F30602-81-C-0132 and F30602-84-C-0171. We would especially like to thank Tom Lawrence and Dick Metzger of RADC for their support and encouragement. It was only through the effort of a large number of people past and present that we have brought the Cronus concept to its current state. These people are too numerous to mention. However, two people no longer associated with the project, William MacGregor, who was part of the original Cronus design team, and Steve Toner, who was a key member of the original implementation team were especially instrumental in getting Cronus off the ground. Finally, we would like to acknowledge the many people from a variety of organizations who contributed to our learning experiences with NSW, and DARPA, which supported much of the earlier distributed operating system work that led to Cronus.

References

[5SOSP] B. Cosell, P. Johnson, J. Malman, R. Schantz, J. Sussman, R. Thomas, and D. Walden, "An Operating System for Computer Resource Sharing," Proc. Fifth Symposium on Operating Systems Principles, Operating Systems Review, vol. 9, no. 5, November 1975.

[Accent] R. Rashid and G. Robertson, "Accent: A communication oriented network operating system kernel," Proc. Eighth Symposium on Operating Systems Principles, Dec. 1981.

[Almes] G. Almes, A. Black, E. Lazowska, and J. Noe, "The Eden system: A technical review," *IEEE Trans. on Software Engineering*, vol. SE-11, no. 1, Jan. 1985, pp. 43-59.

[Diamond] R. Thomas, H. Forsdick, T. Crowley, G. Robertson, R. Schaaf, R. Tomlinson, and V. Travers, "Diamond: A Multimedia Message System Built Upon a Distributed Architecture," submitted for publication.

[ELAN] R. Thomas, R. Schantz, H. Forsdick, "Network Operating Systems, RADC Technical Report TR-78-117, May 1978.

[Grapevine] A. Birrell, R. Levin, R. Needham, M. Schroeder, "Grapevine: An Exercise in Distributed Computing". CACM, Vol. 25, Number 4, April 1982.

[Gurwitz] R. Gurwitz, M. Dean and R. Schantz, "Programming Support in the Cronus Distributed Operating System," in process.

[Jones] A. Jones, "The object model: A conceptual tool for structuring software," in *Lecture Notes in Computer Science, Vol. 60.* Berlin: Springer-Verlag, 1978.

[NSW] H. Forsdick, R. Schantz, and R. Thomas, "Operating Systems for Computer Networks," *Computer*, January 1978.

[NSW1] R. Schantz, R. Thomas, "A Technical Overview of the National Software Works", RADC Technical Report TR-83-80, March 1983.

[Postel] J. Postel, "Internet Control Message protocol - DARPA Internet Program Program Protocol Specfication," *RFC 792* Los Angeles: USC/Informational Sciences Institute, Sept. 1981.

[Robertson] G. Robertson, "The CFS File System," CMU Technical Report, 1982.

[RSEXEC] R.H. Thomas, "A Resource Sharing Executive for the ARPANET", AFIPS Conference Proceedings, vol. 42, June 1973.

[SBS] E. Akkoganlu, A. Bernstein, and R. Schantz, "Interprocess Communication Facilities for Network Operating Systems," *Computer*, June 1974.

[Watson] R. W. Watson, "Distributed System Architecture Model," in *Lecture Notes in Computer Science, Vol. 105*. Berlin: Springer-Verlag, 1980.

[White] J. White "A High Level Framework for Network-Based Resource Sharing", Proc. AFIPS Conference, Vol. 45, 1976.

Appendix B

Robert F. Gurwitz
Michael A. Dean
Richard E. Schantz

BBN Laboratories Incorporated
10 Moulton Street
Cambridge, Massachusetts 02238

ABSTRACT

Technology has made the development of distributed applications more attractive. We need a software environment and tools that will support the development of these applications. The Cronus Distributed Operating System provides both, through its object orientation and through provision of tools to aid in the software development process. We describe our approach to the problem of distributed application development, describe the features of Cronus that support this development, and illustrate how Cronus facilitates development using a Cronus object manager as an example.

Cronus development has been supported by the Rome Air Development Center, under contracts F30602-81-C-0132 and F30602-84-C-0171.

Programming Support in the Cronus Distributed Operating System

ABSTRACT

Technology has made the development of distributed applications more attractive. We need a software environment and tools that will support the development of these applications. The Cronus Distributed Operating System provides both, through its object orientation and through provision of tools to aid in the software development process. We describe our approach to the problem of distributed application development, describe the features of Cronus that support this development, and illustrate how Cronus facilitates development using a Cronus object manager as an example.

1. Introduction

Technology has made the development of distributed applications more attractive, with the proliferation of high-speed local area networks and relatively low-cost personal computers and workstations. These applications may range from simple programs that merely require convenient reference to remote data, to collections of complex algorithms tailored to take advantage of a distributed architecture. A fundamental problem in the development of these applications is complexity, making them difficult to design and implement. Complexity in distributed systems manifests itself in several ways, including the number of components involved, their patterns of communication, and the fact that they may need to be implemented on heterogeneous machines. In the Cronus Distributed Operating System we have attempted to solve these problems in three ways: by adopting a uniform and consistent programming model based on objects, by helping to automate parts of the software development process, and by providing a set of tools to aid in the development of distributed applications. This paper describes these approaches in detail and illustrates how they can be applied to solving the problems of designing and implementing distributed applications by presenting the development of a Cronus object manager as an example.

Cronus is a full-feature distributed operating system that interconnects *clusters* of heterogeneous computers on high-speed local area networks. Its goals are to provide traditional operating system functionality in a distributed computing environment and to serve as a base for developing distributed applications. Some of the characteristic properties of Cronus include survivability of functions, scalability and global management of resources, the ability to substitute components for each other, and convenient operation. The overall system architecture and communication support are described in a companion paper [Schantz].

The remainder of this paper is organized as follows: section 2 discusses in more detail the issues involved in supporting the design and implementation of distributed applications and our approaches to the problem; section 3 describes the features of Cronus that support distributed application development and section 4 gives a specific example of building a simple but representative distributed application. Concluding sections relate our experience with Cronus to date, contrast our approaches with those of related efforts, and outline future plans.

2. Approaches to Distributed Applications Design

2.1. Object Model

The foundation of Cronus is the object model [Jones]. It provides the framework for both the system itself and for application subsystems. System resources (processes, files, devices, etc.) are viewed as abstract objects under the control of a manager process on some host in the Cronus cluster. Every object is an instance of some abstract type, which defines the operations that may be invoked on it. All operations on an object (hence all access to objects) are implemented through the object manager. All the details of the operation's implementation are determined by its manager. To deal with operations where no existing object is involved, such as *create*, Cronus also has the notion of generic objects. Generic objects are roughly equivalent to classes in Smalltalk [Goldberg], the exception being that not all objects of a particular Cronus type are necessarily managed by the same process. Multiple managers of a given object type commonly reside on different hosts in a cluster. This forms the basis of support for global management of resources in Cronus; through manager to manager communication and cooperation. Binding of a manager to an object for a particular operation is accomplished dynamically, through an object location mechanism. Operations on objects may include parameters and may return results.

The object model forms the basis for:

- *Communication.* All Cronus interprocess communication (IPC) is cast in the form of operation invocations and replies. Invocations may be synchronous (waiting for a reply) or asynchronous, and may have one or many targets (using broadcast or multicast). Cronus IPC is built on a set of message-passing primitives and standard protocols for services such as reliable data transport.

- *Location transparency.* The manager for a referenced object may be located anywhere in the cluster, and will be found in a uniform manner by the communications primitives. Cronus objects are named by numeric unique identifiers that are valid anywhere and may have symbolic names associated with them through a distributed hierarchical catalog.

- *Uniformity and coherence.* Cronus applications use the same mechanisms as "system" components. Applications in Cronus may be structured as clients that use application-defined objects and managers. Managers themselves may use the services of other managers of different object types. Managers of the same object type may also cooperate to support more complex abstract types (for example, file managers on different hosts may interact to implement replication through a replicated file object).

- *Access control and authentication.* Access control is provided by the managers on a per-operation basis. Since managers provide the only access to objects, through operations, the object model provides a natural way of introducing access control.

Note that the object model, in Cronus, is a superset of the traditional client/server process model. The client/server model often implies stateless transactions, or if state is maintained, it is kept on a per-connection or per-process basis. Objects provide a convenient mechanism for retaining and managing any state information (called *instance variables*) that exists beyond the lifetime of an individual connection between a client and a server. Similarly, communication in Cronus is a superset of the Remote Procedure

Call (RPC) style of communication. While operation invocation and replies can be cast as synchronous procedure calls, they are not limited to RPC semantics. For example, Cronus IPC supports asynchronous invocations and one-to-many semantics, as well as the one-to-one semantics of RPC.

The object model attacks the complexity problem by providing a framework in which to structure new applications. Probably the most important feature in this framework is that types and operations can be viewed at the appropriate level of abstraction, unconstrained by data representations or other implementation details. Objects will generally correspond directly to real world entities whose behavior is reasonably well understood. Design at this level then becomes primarily a mapping process.

It is also very important that the implementation of an object is completely encapsulated by its managers, providing a clean separation between specification and implementation. This makes it easy for implementations to change as development progresses, requirements change, or new resources become available. In particular, it is very easy to build a simple Cronus manager around an existing non-distributed resource (such as a special device or database), thereby opening access for a network of users.

In Cronus, a distributed application consists of both clients and managers. The same support mechanisms that the object model provides for building managers also apply to client processes. In fact, the distinction between clients and managers is not great, since managers may interact with other managers as clients. We can then structure complex, distributed applications by combining clients and managers that can themselves be, conceptually, relatively simple.

2.2. Other Issues

Several other aspects of distributed applications development add to their complexity but are not directly addressed by the object model. These include programming language support, portability, and the ability to develop applications for heterogeneous computers which may have different internal data representations.

Many efforts to develop distributed systems have concentrated only on homogeneous environments [LeBlanc]. Cronus supports heterogeneity directly by providing extensible standard encodements for different data types and structures, a layer of protocol in the Cronus IPC for exchanging encoded data in messages, and a set of library routines for manipulating encoded messages.

Cronus is designed to be independent of any specific programming language, though most of our development has been done in C. This approach is in contrast to other projects, such as Argus [Liskov] and Eden [Almes], which have taken a more specifically language-centered approach to distributed systems development. The features of Cronus are accessed through subroutine library calls and automatically generated code stubs. Support for multiple languages and simple interfaces to Cronus facilities provide the developer with a familiar and comfortable environment. In large, complex applications, different subparts may be more naturally implemented using different "specialized" languages and hardware architectures (e.g., LISP machines, Fortran, array processors, multi-processors, supercomputers, etc.).

Cronus has been designed for portability, to allow easy integration of new hosts. A Cronus implementation consists of a kernel which implements the low level message passing communication support and object-oriented IPC mechanisms; a set of "system" object managers, which provides basic operating system functionality, such as files, access to I/O devices, etc; and a set of support tools to assist in the creation of both clients and new object managers, including library routines for manipulating encoded messages and for general programming tasks like queue management, string manipulation, encryption, cacheing, etc. Thus, to implement Cronus on a new host, one need only port the kernel and the support library. Since all of these components are written in a higher level language (C) and facilities for handling heterogeneity are included, porting Cronus to new machines is fairly easy. In fact, the approach we have taken in implementing Cronus initially has been to run it as a client of the host's native, or constituent, operating system, though varying degrees of integration are possible. The only requirement for doing this is that the constituent operating system have support for multiple processes and low level network transport facilities (e.g., TCP/IP).

In summary, Cronus provides support for distributed applications through its use of the object model as a tool for functional decomposition and through the resulting client/manager model of interaction. It provides easy-to-use tools and customizable components that can be used to structure new applications. Its support for heterogeneity and multiple languages all help to reduce the complexity of this task. In the following sections, we will show how these approaches relate to the design and implementation of distributed applications.

3. Implementation Support Layers

The Cronus kernel consists of support for the object model through the object-oriented interprocess communication facility and provision of processes for implementing Cronus object managers. The Cronus IPC can be viewed as a layered system of protocols, including link, network, and transport layers implemented with standard protocols (Ethernet, IP, UDP, and TCP are currently used, other protocols of similar functionality could be easily substituted). Above these low-level communications layers are Cronus-specific components that support standard data exchange formats for supporting heterogeneity and a standardized interface for invoking operations on objects.

IPC in Cronus is built on process-to-process messages that form the basis for all Cronus operations. In its simplest form, a Cronus operation consists of a request message sent from a client to an object. and a reply message from the manager of the object sent back to the client. More complex operations may involve several request/reply interactions among various managers. The primitive operations are *send, invoke,* and *receive*. Send and receive are used to exchange messages between processes. Invoke is used to perform an operation on an object. An invoke is delivered directly to the manager of a given object. For more details on the architecture and implementation of Cronus IPC, see [Schantz].

The message encodement layer is independent of the Cronus IPC and provides a canonical representation for data exchanged by applications anywhere in the Cronus cluster. Message data is encoded as a list of key/type/value triples. A key is associated with each value to reference it. Encoding and decoding procedures for each data type depend on the local host's internal data representation. External representations are defined for many common data types (e.g., integers and character strings) and aggregates (arrays and structures). The mechanism is extensible to create user-defined or system-

specific data types. For example, Cronus unique identifiers and access control lists are represented by new data types. Message data is order-independent; the only requirement is that the keys be unique. The order-independent design allows messages to be modified and resent without data reformatting. The programmer's interface to this layer is implemented as a set of library routines for message manipulation, known as the Message Structure Library (MSL).

At the highest level, Cronus defines a set of standards for sending requests and replies between clients and object managers. These standards allow interactions between clients and managers to be structured as either synchronous or asynchronous. This layer, called the Operation Protocol (OP), provides transaction identification, request, reply, and in-progress control and status messages.

As part of Cronus's support for developing object managers, there are also a set of standards for implementing an object manager. These include common object operations (*create, remove,* etc.), which support system-wide functions. For example, each manager is required to implement the *report status* operation and return some abstract notion of status of the resource it manages. This status report is used by a Monitoring and Control Subsystem to provide overall system monitoring of such statistics as processor load, file capacity, etc. Other common operations include those to manipulate access control lists. In addition, a tasking package is provided to allow managers to process requests asynchronously. This is important, since managers may call on other managers for services in the course of performing an operation and one manager may need to suspend processing of a request until a reply is received from the other manager. During this time, it is necessary to allow other client requests to be processed.

This set of standards for implementing Cronus object managers is an important part of Cronus applications support. First, since developing application-defined objects is central to the Cronus philosophy of design, aids to the construction of new managers are a necessary tool for the programmer. The standards for manager implementation form an interface between the "system" and the application--one that can be crucial to the resulting form of the application code and one that has an significant effect on performance. Second, as the standards for new manager construction become better defined, there are opportunities for automating the generation of the common parts of new managers, hence reducing the complexity and tedium of the task of coding them. We believe that automated manager development is a useful new technique for distributed applications development; we will describe this approach in more detail in the next section.

4. Building Applications

The task of developing Cronus applications consists of implementing clients which access existing managers, and usually includes building one or more managers of application-specific objects. The principal goal of automating the development of managers is to reduce the amount of code that the developer must write which is not directly related to the application. The developer is then freed to implement only those operations that pertain to his objects; all existing system services are easily accessible without additional programming. The automated development technique also produces subroutine-style interfaces for invoking operations on objects that can be used by clients.

Cronus object managers have a common framework that includes the overall control structure and handling of parameter and reply messages. Our approach is to take non-procedural specifications of object types and their operations (a message protocol) and to use a program to generate the internal data structures, message parsing routines, and interfaces to the Cronus run-time environment. The application developer provides the domain-specific operation processing routines that perform the actual function of the manager. The generated code can be thought of as the glue between the developer's routines and Cronus run-time support.

The automatically generated code includes facilities for operation-specific message parsing and argument validation, conversion of arguments and instance variables to internal data formats that are easy to manipulate, and the subroutine-style interface encapsulating operation invocations for clients. These subroutine calls provide client access to the object managers through a simple, procedural interface.

The run-time support includes facilities for request dispatching, access control checks, concurrent servicing of multiple requests, and long-term storage management of object instance variables on disk. This provides much of the overall control structure for the manager and assists the developer in managing frequently performed tasks that can be hard to implement. For example, maintaining instance variables is difficult in practice, due to the complexities of format conversion and secondary storage management.

The manager generation process is implemented in two phases. The first parses object type and operation descriptions and stores them in an intermediate form in the *protocol database*. This database is used by several back-end code generators in the second phase, which actually produces the code. The output of the second phase includes the manager and client components described above.

4.1. Example: Software Distribution Manager

Throughout this section, we will refer to the Cronus Software Distribution Manager as an example. This manager was built to assist in maintaining our system code, which currently runs in five different constituent operating system environments on about 15 machines. A *package* consists of a collection of source or executable files to be maintained at a collection of sites. One of these sites is designated primary; updates applied to the file copies there will be propagated to those at other sites by the *distribute* operation.

Though simple, we believe this example to be representative of potential applications in several ways. First, the package (representing an individual software component) is an abstract entity at an appropriate level to be dealt with by the user. Implementation details for the abstraction are hidden within the manager, which totally controls coordination of the software distribution process. Second, each object has a complex (variable length, multi-component) data structure associated with it. The management of such data structures should be aided by the run-time support. Finally, the manager will almost certainly evolve to incorporate additional facets of the software distribution process. It should be easy to incorporate new operations and changes to existing ones into the manager.

4.2. Protocol Specification

The first and most important step in the manager development process is the specification of the message protocols for the new object types. This totally defines the interface visible to clients. The specification is then processed by a program that parses it and stores it in an intermediate form for use in the code generation phase. A (slightly) abbreviated definition for the Cronus type *Package* is given in

```
type Package
        subtype of Object
        rights are modifyfiles, modifysites, distribute;

generic operation Create (
        Key_Description: ASC;
        Key_PrimaryHost: EHOST;
        Key_PrimaryDirectory: ASC;
        optional Key_IACLHints: array of EUID)

        returns (Key_ObjectUID: EUID)
        requires create;

operation AddFile (Key_FileName: ASC)
        requires modifyfiles;

operation RemoveFile (Key_FileName: ASC)
        requires modifyfiles;

operation Distribute ()
        returns (Key_NumberFilesUpdated: U16I)
        requires distribute;

variable cantype PKGVARIABLE
        representation is PkgVariable:
        record
                description: ASC;
                primarycopy: SITEDATA;
                files: array of ASC;
                sites: array of SITEDATA;
end PKGVARIABLE;

end type Package;
```

Figure 1.

The specification can be divided into three parts. First is the definition of the type, *Package*. Its parent in the type hierarchy, from which it inherits operations and access control information, is specified (**subtype of...**). In this case, the *Package* type's parent is *Object*, the root of the hierarchy, where operations common to all Cronus objects are defined. These include the basic *create* and *remove* operations, as well as those used to support generic services, such as monitoring and control and access

control. Children in the hierarchy are also free to augment and redefine the features they inherit. Also included in the type definition are the object's application-specific abstract access control rights (**rights are...**) which are later bound to specific operations. Rights are also inherited from the parent type.

The second component of the specification is the definition of the operations that implement the object type (**operation...**). Each operation may take arguments and is assumed to return a reply message (**returns...**). For those operations where no reply is specified, a successful reply is sent by default when the operation completes. The argument and reply messages are all specified in terms of keys and data types defined by the message structuring layer. Also included are the access rights required for the performance of the operation (**requires...**).

The third part of the specification is the definition of a new canonical data type that represents the instance variables, or retained state, of the object that the manager must maintain (**variable cantype...**). The new data type can be used for exchange between hosts over the network, and provides a convenient format for storage and transmission of a complex data structure as a linear sequence of bytes.

4.3. Operation Scenario

Once the protocols have been defined and processed, a program is run to generate the manager code from the intermediate form produced in the specification phase. Its outputs include tables to drive the dispatching/tasking mechanism, functions to parse and validate messages, type conversion routines, and client subroutine interfaces for operation invocation. This section traces the processing of an operation, from the client's invocation through the manager's reply.

When a client invokes an operation on an object, the named object is located and the invoke message is sent to its manager by the Cronus kernel. This message includes the type, operation code, and any arguments to the operation. The manager receives the message and creates a new task to service it. It then verifies the operation code from the message and, assuming the operation code is legitimate, retrieves a descriptor for the object from its private store (the *object database*). Not finding the descriptor means that the object does not exist, and an error reply is returned to the client. The descriptor includes the object's access control list as well as application-dependent data.

An access control check is then made. Assuming access is allowed, the operation arguments are converted into an internal representation by an automatically generated parsing routine that is specific to the operation. This representation is dependent upon the language chosen to implement the processing routines. The C structure for the arguments to the *Create* operation defined above is given in Figure 2, which shows the arguments in their internal representation and flags for the presence of optional arguments and the dimensionality of arrays. The parsing routines also verify the presence of required arguments, and perform type checking.

If the parse is successful, processing of the operation begins. A buffer for the reply message is allocated, and the operation processing routine is called. This is the first time the application developer's code is executed. The actual amount of code written by the implementor to process an operation may be quite small. The complete code to implement the *RemoveFile* operation for the *Package* object is given in

```
struct argpkgCreate
{
        /* arguments */

    char *Description;
    HOSTNUM PrimaryHost;
    char *PrimaryDirectory;
    UID *IACLHints;

    struct
    {
        BOOL IACLHints;
    } specified;  /* optional argument flags */

    struct
    {
        int IACLHints;
    } dimensions;  /* array dimesionality */
};

pkgRemoveFile (request, args)
struct RequestParms *request;   /* generic header, object descriptor */
struct argpkgRemoveFile *args;/* operation arguments */
{
        PkgVariable *pkgdes;
        int i, j;

        pkgdes = (PkgVariable *)
                getvardata (PKGVARIABLE, request->rp_objdes);  /* get instance variables */

        for (i = 0; i < pkgdes->nfiles; i++)
                if (scmp (args->FileName, pkgdes->files[i]) == 0)
                {
                        for (j = i-1; j < pkgdes->nfiles; j++)        /* delete entry */
                                pkgdes->files[j-1] = pkgdes->files[j];
                        pkgdes->nfiles--;
                }

        putvardata (PKGVARIABLE, request->rp_objdes, pkgdes);  /* update instance variables */
}
```

Figure 3.

There are several observations to make about the software distribution manager example. First, the definition of the object type is at a high level. Here, we consider such issues as abstract access rights, operation arguments and replies, key names, and canonical data types. The details of the internal data structure definitions, message formats, and the control structure of the manager are all generated

automatically. Second, many of the run-time details, such as operation validation, access control checks, data conversion, type checking, task dispatching, and error handling, are also done automatically by the run-time support. Third, and most important, the actual amount of code written by the application developer is small and concentrates almost exclusively on accomplishing the specific task to be performed by the manager. Thus, the programmer is free to concentrate on his problem, rather than on the details of Cronus.

4.4. Implementation

As described above, automated manager generation consists of parsing the type specification and producing the code. In addition to the manager and client code, the process results in tables for a command level operation invoking tool which has proved extremely useful for debugging and for the implementation of simple commands. Documentation suitable for user manuals is also generated in the form of the message protocol specification (similar to Figure 1, but in a slightly cleaner format for ease of understanding). Annotations may also be added to the protocol database to be generated as part of the specification documentation [Sands].

The generated code includes five files: the dispatch tables for the operations, the parse routines for the operation arguments, a header file that contains the arguments' internal structure definition (as in Figure 2), the conversion routines for any canonical types that have been defined, and the client subroutine interfaces for operation invocations. The programmer then supplies the specific operation processing routines (as in Figure 3). To produce the manager, the first four files are compiled and linked together with this supplied code.

Note that the programmer must know the naming conventions for the generated data structures that are referred to in his code. Alternatively, a preprocessor could be used to translate operation invocations in the client code to the proper Cronus run-time calls. This would have two disadvantages. First, a new preprocessor would have to be supplied for each programming language. Second, the specification would have to be re-processed with any change in the operation processing routines. This could be time-consuming for complex specifications.

5. Conclusion

5.1. Experience

Cronus has been in limited operational use at BBN for a year. It has been used to support its own software development. The BBN configuration consists of 15 hosts, including DEC VAXes running both 4.2BSD UNIX and VMS constituent operating systems, SUN Workstations, BBNCC C70 UNIX systems, and several single-function Motorola 68000 microprocessor systems, called Generic Computing Elements (used as file servers, authentication servers, and terminal access controllers). An additional cluster is planned for installation in the near future at the Rome Air Development Center.

All of our current software is written in C, though some earlier implementation was done in Pascal. We do not anticipate major problems in porting it to new languages (including retargeting of the manager code generator). Most of our software is portable to all five of our current execution environments. We have found it easy to relocate services as new machines and operating systems are integrated into a Cronus cluster.

A number of Cronus managers have been built to date. Initial development has centered on managers which provide standard operating system services as building blocks for applications, including several types of file managers, the catalog manager, process manager, and authentication manager. The automated development software has been available for about five months. During this time it has been used to build the authentication manager, the constituent operating system interface manager (which allows constituent host files and directories to be manipulated as Cronus objects), the logging component of the monitoring and control subsystem, and the software distribution manager. We plan to convert other managers to use it in the near future.

The first application of the automated development technique was a rewrite of the authentication manager. The new manager consists of about 1500 lines of programmer-specified C code, plus 1000 lines of generated code and about 500 lines of library code implementing the common operations. Its predecessor consisted of about 6500 lines of hand-coded Pascal. Currently, the processing of the specification takes about one minute of elapsed time on a VAX 11/780 for the authentication manager, which is typical. The code generation phase takes about 1.5 minutes. It is worth noting that the new manager was done by one person in about two weeks, compared to several months for the original. The new authentication manager has been in production use ever since and exhibits performance similar to the original manager implementation.

5.2. Related Work

The object model has been successfully employed in a number of systems, most notably Smalltalk [Goldberg] and Hydra [Wulf]. We think the structure imposed (and opportunities for software leverage thereby afforded) makes it easier to design and develop distributed applications than in systems based on unconstrained communications between processes, such as the V Kernel [Cheriton]. We also think the abstraction capabilities provided make it superior to a simple distributed filesystem approach, such as Locus [Walker], for many applications.

With respect to software production, our facilities are most likely to be contrasted with language-based systems such as Argus [Liskov] and Eden [Almes], and RPC-based approaches such as Cedar [Birrell]. None of these have dealt with issues of heterogeneity (of hardware or software environments), which is a cornerstone of the Cronus approach. We also feel that preserving as much as possible of the programmer's existing environment leads to faster development and greater integration of existing resources.

5.3. Future Plans

We are pursuing new Cronus development in several areas. These include continuing to evolve and apply our automated production technique to new managers and applications areas: investigating strategies for resource management, survivability through resource replication, and transaction management; and integrating new architectures, such as multiprocessors and symbolic processors. Our efforts in supporting Cronus for operational use include installation of additional clusters at other sites, and work in performance measurement and enhancement. Finally, we believe that the best way to evaluate and evolve a system is to use it. This process is taking place through both our own use of Cronus for continued software development and its use in building command and control applications [Berets].

Acknowledgments

We wish to acknowledge the ideas and energy of our colleagues at BBN Laboratories and elsewhere who contributed to the design and implementation of Cronus; in particular, for the applications support aspects we thank Dr. William MacGregor, Robert Walsh, Richard Sands, and Dr. Benjamin Woznick. We also acknowledge our sponsors at the Rome Air Development Center, Thomas Lawrence and Richard Metzger, who provided support and assistance for the development of Cronus.

References

[Almes] G. Almes, A. Black, E. Lazowska, and J. Noe, "The Eden system: A technical review," *IEEE Trans. on Software Engineering,* vol. SE-11, no. 1, Jan. 1985, pp. 43-59.

[Berets] J. Berets, R. Mucci, and R. Schantz, "Cronus: A Testbed for Developing Distributed Systems," in process.

[Birrell] A. Birrell and B. Nelson, "Implementing remote procedure calls," *ACM Trans. on Computer Systems,* vol. 2, no. 1, Feb. 1984, pp. 39-59.

[Cheriton] D. Cheriton and W. Zwaenepoel, "The distributed V kernel and its performance for diskless workstations." *Proc. 9th Symposium on Operating Systems Principles,* ACM, Oct. 1983. pp. 129-140.

[Goldberg] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation.* Reading, MA: Addison-Wesley, 1983.

[Jones] A. Jones, "The object model: A conceptual tool for structuring software," in *Lecture Notes in Computer Science, Vol. 60.* Berlin: Springer-Verlag, 1978.

[LeBlanc] T. LeBlanc, R. Gerber, and R. Cook. "The StarMod distributed programming kernel," *Software Practice and Experience,* vol. 14, no. 12, Dec. 1984, pp. 1123-1139.

[Liskov] B. Liskov, "On linguistic support for distributed programs," *IEEE Trans. on Software*

Engineering, vol. SE-8, May 1982, pp. 203-210.

[Sands] R. Sands, ed., "The Cronus User's Manual," BBN Technical Report.

[Schantz] R. Schantz, R. Thomas, G. Bono, "The Architecture and Implementation of the Cronus Distributed Operating System," in process.

[Walker] B. Walker, G. Popek, R. English, C. Kline, and G. Theil, "The LOCUS distributed operating system," in *Proc. 9th Symposium on Operating Systems Principles,* ACM, Oct. 1983, pp. 49-69.

[Wulf] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The kernel of a multiprocessor operating system," *Comm. ACM*, vol. 17, no. 6, June 1974, pp. 337-345.

Appendix C

RFC 947: Multi-network Broadcasting within the Internet

Network Working Group
Request for Comments: 947

Ken Lebowitz
David Mankins
BBN Laboratories

Multi-network Broadcasting within the Internet

1. Status of this Memo

This RFC describes the extension of a network's broadcast domain to include more than one physical network through the use of a broadcast packet repeater.

The following paper will present the problem of multi-network broadcasting and our motivation for solving this problem which is in the context of developing a distributed operating system. We discuss different solutions to extending a broadcast domain and why we chose the one that has been implemented. In addition, there is information on the implementation itself and some notes on its performance.

It is hoped that the ideas presented here will help people in the Internet who have applications which make use of broadcasting and have come up against the limitation of only being able to broadcast within a single network.

The information presented here is accurate as of the date of publication but specific details, particularly those regarding our implementation, may change in the future. Distribution of this memo is unlimited.

2. The Problem

Communication between hosts on separate networks has been addressed largely through the use of Internet protocols and gateways. One aspect of internetwork communication that hasn't been solved in the Internet is extending broadcasting to encompass two or more networks. Broadcasting is an efficient way to send information to many hosts while only having to transmit a single packet. Many of the current local area network (LAN) architectures directly support a broadcast mechanism. Unfortunately, this broadcast mechanism has a shortcoming when it is used in networking environments which include multiple LANs connected by gateways such as in the DARPA Internet. This shortcoming is that broadcasted packets are only received by hosts on the physical network on which the packet was broadcast. As a result, any application which takes advantage of LAN broadcasting can only broadcast to those hosts on its physical network.

We took advantage of broadcasting in developing the Cronus Distributed Operating System [1]. Cronus provides services and communication to processes distributed among a variety of different

types of computer systems. Cronus is built around logical clusters of hosts connected to one or more high-speed LANs. Communication in Cronus is built upon the TCP and UDP protocols. Cronus makes use of broadcasting for dynamically locating resources on other hosts and collecting status information from a collection of servers. Since Cronus's broadcast capabilities are not intended to be limited to the boundaries of a single LAN, we needed to find some way to extend our broadcasting domain to include hosts on distant LANs in order to experiment with clusters that span several physical networks. Cronus predominantly uses broadcasting to communicate with a subset of the hosts that actually receive the broadcasted message. A multicast mechanism would be more appropriate, but was unavailable in some of our network implementations, so we chose broadcast for the initial implementation of Cronus utilities.

3. Our Solution

The technique we chose to experiment with the multi-network broadcasting problem can be described as a "broadcast repeater". A broadcast repeater is a mechanism which transparently relays broadcast packets from one LAN to another, and may also forward broadcast packets to hosts on a network which doesn't support broadcasting at the link-level. This mechanism provides flexibility while still taking advantage of the convenience of LAN broadcasts.

Our broadcast repeater is a process on a network host which listens for broadcast packets. These packets are picked up and retransmitted, using a simple repeater-to-repeater protocol, to one or more repeaters that are connected to distant LANs. The repeater on the receiving end will rebroadcast the packet on its LAN, retaining the original packet's source address. The broadcast repeater can be made very intelligent in its selection of messages to be forwarded. We currently have the repeater forward only broadcast messages sent using the UDP ports used by Cronus, but messages may be selected using any field in the UDP or IP headers, or all IP-level broadcast messages may be forwarded.

4. Alternatives to the Broadcast Repeater

We explored a few alternatives before deciding on our technique to forward broadcast messages. One of these methods was to put additional functions into the Internet gateways. Gateways could listen at the link-level for broadcast packets and relay the packets to one or more gateways on distant LANs. These gateways could then transmit the same packet onto their networks using the local network's link-level broadcast capability, if one is available. All gateways participating in this scheme would have to maintain tables

of all other gateways which are to receive broadcasts. If the recipient gateway was serving a network without a capacity to broadcast it could forward the messages directly to one or more designated hosts on its network but, again, it would require that tables be kept in the gateway. Putting this sort of function into gateways was rejected for a number of reasons: (a) it would require extensions to the gateway control protocol to allow updating the lists gateways would have to maintain, (b) since not all messages (e.g., LAN address- resolution messages) need be forwarded, the need to control forwarding should be under the control of higher levels of the protocol than may be available to the gateways, (c) Cronus could be put into environments where the gateways may be provided by alternative vendors who may not implement broadcast propagation, (d) as a part of the underlying network, qateways are likely to be controlled by a different aqency from that controlling the configuration of a Cronus system, adding bureaucratic complexity to reconfiguration.

Another idea which was rejected was to put broadcast functionality into the Cronus kernel. The Cronus kernel is a process which runs on each host participating in Cronus, and has the task of routing all messages passed between Cronus processes. The Cronus kernel is the only program in the Cronus system which directly uses broadcast capability (other parts of Cronus communicate using mechanisms provided by the kernel). We could either entirely remove the Cronus kernel's dependence on broadcast, or add a mechanism for emulating broadcast using serially-transmitted messages when the underlying network does not provide a broadcast facility itself. Either solution requires all Cronus kernel processes to know the addresses of all other participants in a Cronus system, which we view as an undesirable limit on configuration flexibility. Also, this solution would be Cronus-specific, while the broadcast-repeater solution is applicable to other broadcast-based protocols.

5. Implementation

The broadcast repeater is implemented as two separate processes - the forwarder and the repeater. The forwarder process waits for broadcast UDP packets to come across its local network which match one or more specific port numbers (or destination addresses). When such a packet is found, it is encapsulated in a forwarder-repeater message sent to a repeater process on a foreign network. The repeater then relays the forwarded packet onto its LAN using that network's link-level broadcast address in the packet's destination field, but preserving the source address from the original packet.

When the forwarder process starts for the first time it reads a

configuration file. This file specifies the addresses of repeater processes, and selects which packets should be forwarded to each repeater process (different repeaters may select different sets of UDP packets). The forwarder attempts to establish a TCP connection to each repeater listed in the configuration file. If a TCP link to a repeater fails, the forwarder will periodically retry connecting to it. Non-repeater hosts may also be listed in the configuration file. For these hosts the forwarder will simply replace the destination broadcast address in the UDP packet with the host's address and send this new datagram directly to the non-repeater host.

If a repeater and a forwarder co-exist on the same LAN a problem may arise if the forwarder picks up packets which have been rebroadcast by the repeater. As a precaution against rebroadcast of forwarded packets ("feedback" or "ringing"), the forwarder does not connect to any repeaters listed in its configuration file which are on the same network as the forwarder itself. Also, to avoid a broadcast loop involving two LANs, each with a forwarder talking to a repeater on the other LAN, forwarders do not forward packets whose source address is not on the forwarder's LAN.

6. Experience

To date, the broadcast repeater has been implemented on the VAX running 4.2 BSD UNIX operating system with BBN's networking software and has proven to work quite well for our purposes. Our current configuration includes two Ethernets which are physically separated by two other LANs. For the past few months the broadcast repeater has successfully extended our broadcast domain to include both Ethernets even though messages between the two networks must pass through at least two gateways. We were forced to add a special capability to the BBN TCP/IP implementation which allows privileged processes to send out IP packets with another host's source address.

The repeater imposes a fair amount of overhead on the shared hosts that currently support it due to the necessity of waking the forwarder process on all UDP packets which arrive at the host, since the decision to reject a packet is made by user-level software, rather than in the network protocol drivers. One solution to this problem would be to implement the packet filtering in the system kernel (leaving the configuration management and rebroadcast mechanism in user code) as has been done by Stanford/CMU in a UNIX packet filter they have developed. As an alternative we are planning to rehost the implementation of the repeater function as a specialized network service provided by a microcomputer based

real-time system which is already part of our Cronus configuration. Such a machine is better suited to the task since scheduling overhead is much less for them than it is on a multi-user timesharing system.

7. Reference

[1] "Cronus, A Distributed Operating System: Phase 1 Final Report", R. Schantz, R. Thomas, R. Gurwitz, G. Bono, M. Dean, K. Lebowitz, K. Schroder, M. Barrow and R. Sands, Technical Report No. 5885, Bolt Beranek and Newman, Inc., January 1985. The Cronus project is supported by the Rome Air Development Center.

8. Editors Note

Also see RFCs 919 and 940 on this topic.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.